# Homework 6: Declarative Concurrency

Due: Tuesday, November 14, 2006.

In this homework you will learn basic techniques of declarative concurrent programming, including stream programming and lazy functional programming. Many of the problems below exhibit polymorphism. The problems as a whole illustrate how functional languages work without hidden side-effects. Don't use side effects (assignment and cells) in your solutions.

For all programing tasks, you must run your code using the Mozart/Oz system. For these you must also provide evidence that your program is correct (for example, test cases). For testing, you may want to use tests based on my code in the course library file `Test.oz`, as shown in homework 4.

Hand in a printout of your code and the output of your testing, for all questions that require code.

Be sure to clearly label what problem each area of code solves with a comment.

Don't hesitate to contact the staff if you are stuck at some point.

Read Chapter 4 of the textbook [RH04]. You may also want to read a tutorial on the concepts of functional programming languages, such as Hudak's computing survey article mentioned in the "Introduction to the Literature" handout.

## Streams and Lazy Functional Programming

The following problems, many of which are due to John Hughes, relate to modularization of numeric code using streams and lazy execution. In particular, we will explore the Newton-Raphson algorithm. This algorithm computes better and better approximations to the square root of a floating point number `N` from a previous approximation `X` by using the following curried function.

```
declare
% Next: <fun {$ <Float>}: <fun {$ <Float>}: Float>>
fun {Next N}
   fun {$ X}
      (X + N / X) / 2.0
   end
end
```

In the following we will use the type `<Stream T>` as a synonym for the type `<List T>`, in which the list is presumed to be infinite. Note that `nil` never occurs in a `<Stream T>`.

As an aid to writing code for this section, and for testing that code, we provide library modules for approximate comparisons of floating point numbers and for testing with approximate comparisons. The floating point approximate comparison code is shown in Figure 1 on the following page. The testing code for floating point numbers is shown in Figure 2 on page 3 and Figure 3 on page 4.

1. (10 points) Write a lazy function

   `StreamIterate: <fun lazy {$ <fun {$ T}: T> T}: <Stream T>>`

   such that `{StreamIterate` $F$ $X$`}` takes a function $F$ and a value $X$ and returns the stream

   $$X \mid \{FX\} \mid \{F\{FX\}\} \mid \{F\{F\{FX\}\}\} \mid \ldots,$$

   that is, the stream whose $i^{th}$ item, counting from 1, is $F^{i-1}$ applied to $X$.

   The examples in Figure 4 on page 4 are written using the `WithinTest` procedure from Figure 3 on page 4.

2. (5 points) Why does `StreamIterate` (see Exercise 1) have to be lazy? Give a brief answer.

3. (10 points) Write a function

```
% $Id: FloatComparisons.oz,v 1.4 2006/10/27 08:57:49 leavens Exp $
% Approximate floating point numeric comparison procedures for Oz.
% AUTHOR: Gary T. Leavens

functor $
export
   within: Within
   withinMaker: WithinMaker
   relative: Relative
   relativeMaker: RelativeMaker
   compareLists: CompareLists
   withinLists: WithinLists
   relativeLists: RelativeLists
define
   %% Return true iff the difference between X and Y
   %% is no larger than Epsilon
   fun {Within Epsilon X Y} {Abs X-Y} =< Epsilon end

   %% Partly curried version of Within
   fun {WithinMaker Epsilon} fun {$ X Y} {Within Epsilon X Y} end end

   %% Return true iff the corresponding lists are
   %% equal relative to the given predicate
   fun {CompareLists Pred Xs Ys}
      case Xs#Ys of
         nil#nil then true
      [] (X|Xr)#(Y|Yr) then {Pred X Y} andthen {CompareLists Pred Xr Yr}
      else false
      end
   end

   %% Return true iff the lists are equal
   %% in the sense that the corresponging elements
   %% are equal to within Epsilon
   fun {WithinLists Epsilon Xs Ys}
      {CompareLists {WithinMaker Epsilon} Xs Ys}
   end

   %% Return true iff the ratio of X-Y to Y is within Epsilon
   fun {Relative Epsilon X Y} {Abs X-Y} =< Epsilon*{Abs Y} end

   %% Partly curried version of Relative
   fun {RelativeMaker Epsilon} fun {$ X Y} {Relative Epsilon X Y} end end

   %% Return true iff the lists are equal
   %% in the sense that the corresponging elements
   %% are relatively equal to within Epsilon
   fun {RelativeLists Epsilon Xs Ys}
      {CompareLists {RelativeMaker Epsilon} Xs Ys}
   end
end
```

Figure 1: Comparisons for floating point numbers. This functor is available in the course `lib` directory.

```
% $Id: FloatTesting.oz,v 1.4 2006/10/27 08:57:49 leavens Exp $
% Testing for floating point numbers.
% AUTHOR: Gary T. Leavens

functor $
import
   System(showInfo)
   FloatComparisons
export
   testMaker: TestMaker
   standardTolerance: StandardTolerance
   withinTest: WithinTest
   relativeTest: RelativeTest

define
   %% TestMaker returns a procedure P such that {P Actual '=' Expected}
   %% is true if {FloatCompare Epsilon Actual Expected} (for Floats)
   %% or if {FloatListCompare Epsilon Actual Expected} (for lists of Floats)
   %% If so, print a message, otherwise throw an exception.
   fun {TestMaker FloatCompare FloatListCompare Epsilon}
      fun {Compare Actual Expected}
         if {IsFloat Actual} andthen {IsFloat Expected}
         then {FloatCompare Epsilon Actual Expected}
         elseif {IsList Actual} andthen {IsList Expected}
         then {FloatListCompare Epsilon Actual Expected}
         else false
         end
      end
   in
      proc {$ Actual Connective Expected}
         if {Compare Actual Expected}
         then {System.showInfo
               {Value.toVirtualString Actual 5 10}
               # ' ' # Connective # ' '
               # {Value.toVirtualString Expected 5 10}}
         else {Exception.raiseError
               testFailed(actual:Actual
                          connective:Connective
                          expected:Expected
                          debug:unit)
              }
         end
      end
   end

   StandardTolerance = 1.0e~3
   WithinTest = {TestMaker FloatComparisons.within
                           FloatComparisons.withinLists StandardTolerance}
   RelativeTest = {TestMaker FloatComparisons.relative
                             FloatComparisons.relativeLists StandardTolerance}
end
```

Figure 2: Testing code for floating point. This puts output on standard output (the `*Oz Emulator*` window). The module `FloatComparisons` is shown in Figure 1 on the preceding page. This functor is available in the course `lib` directory.

```
% $Id: FloatTest.oz,v 1.2 2006/10/26 07:19:52 leavens Exp $
% AUTHOR: Gary T. Leavens

declare
local
   [Testing FloatTesting FloatComparisons]
   = {Module.link ['Testing.ozf' 'FloatTesting.ozf' 'FloatComparisons.ozf']}
in
   StartTesting = Testing.start
   Test = Testing.test
   Assert = Testing.assert
   Assume = Testing.assert
   FloatTestMaker = FloatTesting.testMaker
   StandardTolerance = FloatTesting.standardTolerance
   WithinTest = FloatTesting.withinTest
   RelativeTest = FloatTesting.relativeTest
   Within = FloatComparisons.within
   WithinMaker = FloatComparisons.withinMaker
   Relative = FloatComparisons.relative
   RelativeMaker = FloatComparisons.relativeMaker
   CompareLists = FloatComparisons.compareLists
   WithinLists = FloatComparisons.withinLists
   RelativeLists = FloatComparisons.relativeLists
end
```

Figure 3: Testing code for floating point. This works in the Mozart system's Oz Programming Interface. The module linked is shown in Figure 2 on the preceding page. This file is available in the course `lib` directory To use it, copy the files from the course directory to your own directory and then put `\insert 'FloatTest.oz'` in your file.

```
\insert 'StreamIterate.oz'
\insert 'Next.oz'
\insert 'FloatTest.oz'
{StartTesting 'StreamIterate...'}
{WithinTest {List.take {StreamIterate {Next 1.0} 1.0} 7}
 '~=~' [1.0 1.0 1.0 1.0 1.0 1.0 1.0]}
{WithinTest {List.take {StreamIterate {Next 9.0} 1.0} 7}
 '~=~' [1.0 5.0 3.4 3.0235 3.0001 3.0 3.0]}
{WithinTest {List.take {StreamIterate {Next 200.0} 1.0} 7}
 '~=~' [1.0 100.5 51.245 27.574 17.414 14.449 14.145]}
{RelativeTest {List.take {StreamIterate {Next 0.144} 7.0} 9}
 '~=~' [7.0 3.5103 1.7757 0.92838 0.54174 0.40378 0.3802 0.37947 0.37947]}
{RelativeTest {List.take {StreamIterate fun {$ X} X*X end 2.0} 9}
 '~=~' [2.0 4.0 16.0 256.0 65536.0 4.295e009 1.8447e019 3.4028e038
        1.1579e077]}
{RelativeTest {List.take {StreamIterate fun {$ X} X/3.0 end 10.0} 8}
 '~=~' [10.0 3.3333 1.1111 0.37037 0.12346 0.041152 0.013717 0.0045725]}
```

Figure 4: Tests for Exercise 1 on page 1.

```
Approximations: <fun {$ Float Float}: <Stream Float>>
```

such that `{Approximations N A0}` returns the infinite list of approximations to the square root of `N`, starting with `A0`, according to the Newton-Raphson method, that is by iterating `{Next N}`.

4. (5 points) Does your solution to Exercise 3 on page 1 have to be lazy? Briefly explain.

5. (10 points) Write a function

   ```
   ConvergesTo: <fun {$ <Stream T> <fun {$ T T}: Bool>}: T>
   ```

   such that `{ConvergesTo Xs Pred}` looks down the stream `Xs` to find the first two consecutive elements of `Xs` that satisfy `Pred`, and it returns the second of these consecutive elements. (It will never return if there is no such pair of consecutive elements.) Figure 6 on the next page gives some examples.

6. (10 points) Using `FloatComparisons.withinMaker` from Figure 1 on page 2 and the other pieces given above, write a function

   ```
   SquareRoot: <fun {$ Float Float Float}: Float>
   ```

   such that `{SquareRoot A0 Epsilon N}` returns an approximation to the square root of `N` that is within `Epsilon`. The parameter `A0` is used as an initial guess in the Newton-Raphson method. Examples are given in Figure 7 on the following page.

7. (10 points) Using `FloatComparisons.relativeMaker` from Figure 1 on page 2 and the other pieces given above, write a function

   ```
   RelativeSquareRoot: <fun {$ Float Float Float}: Float>
   ```

   such that `{RelativeSquareRoot A0 Epsilon N}` returns an approximation to the square root of `N` that only has a relative error of `Epsilon`. The parameter `A0` is used as an initial guess in the Newton-Raphson method. When iterating, keep going until the ratio of the difference between the last and the previous approximation to the last approximation approaches 0, instead of waiting for the differences between the approximations themselves to approach zero. (This is equivalent to iterating until the ratio of the last two approximations approaches 1.) This is test for convergence is better for square roots of very large numbers, and for square roots of very small numbers. Examples are given in Figure 8 on page 7.

8. (15 points) You may recall that the derivative of a function `F` at a point `X` can be approximated by the following function.

   ```
   declare
   fun {EasyDiff F X Delta} ({F (X+Delta)} - {F X}) / Delta end
   ```

   Good approximations are given by small values of `Delta`, but if `Delta` is too small, then rounding errors may swamp the result. One way to choose `Delta` is to compute a sequence of approximations, starting with a reasonably large one. If `{WithinMaker Epsilon}` is used to select the first approximation that is accurate enough, this can reduce the risk of a rounding error affecting the result.

   Your task is to write a function

   ```
   DiffApproxims: <fun {$ Float <fun {$ Float}: Float> Float}:
                       <Stream Float>>
   ```

   such that `{DiffApproxims Delta0 F X}` returns an infinite list of approximations to the derivative of `F` at `X`, where at each step, the current `Delta` is halved. Examples are given in Figure 9 on page 7.

   Hint: do you need to make something lazy to make this work?

9. (15 points) Write a function

```
\insert 'Approximations.oz'
\insert 'FloatTest.oz'
{StartTesting 'Approximations...'}
{WithinTest {List.take {Approximations 1.0 1.0} 7}
 '~=~' [1.0 1.0 1.0 1.0 1.0 1.0 1.0]}
{RelativeTest {List.take {Approximations 2.0 1.0} 5}
 '~=~' [1.0 1.5 1.41667 1.41422 1.41421]}
{RelativeTest {List.take {Approximations 64.0 1.0} 5}
 '~=~' [1.0 32.5 17.2346 10.474 8.29219]}
```

Figure 5: Tests for Exercise 3 on page 1.

```
\insert 'ConvergesTo.oz'
\insert 'FloatTest.oz'

fun lazy {Repeat X} X|{Repeat X} end

{StartTesting 'ConvergesTo...'}
{WithinTest {ConvergesTo
             {Append [1.0 3.5 4.5] {Repeat 7.0}}
             {WithinMaker 1.01}
            }
 '~=~' 4.5}
{WithinTest {ConvergesTo
             {Append [1.0 32.5 17.2346 10.474 8.29219 8.00515] {Repeat 8.0}}
             {WithinMaker 0.5}
            }
 '~=~' 8.00515}
```

Figure 6: Tests for Exercise 5 on the preceding page.

```
\insert 'SquareRoot.oz'
\insert 'FloatTest.oz'
{StartTesting 'SquareRoot...'}
Millionth = 0.0000001
WithinMillionth = {FloatTestMaker
                   Within
                   WithinLists
                   Millionth}
{WithinMillionth {SquareRoot 1.0 Millionth 2.0} '~=~' 1.41421356}
{WithinMillionth {SquareRoot 1.0 Millionth 4.0} '~=~' 2.0}
{WithinMillionth {SquareRoot 1.0 Millionth 64.0} '~=~' 8.0}
{WithinMillionth {SquareRoot 1.0 Millionth 3.14159} '~=~' 1.7724531}
```

Figure 7: Tests for Exercise 6 on the previous page.

```
\insert 'RelativeSquareRoot.oz'
\insert 'FloatTest.oz'
{StartTesting 'RelativeSquareRoot...'}
TenThousandth = 0.000001
RelativeTenThousandth = {FloatTestMaker
                  Relative
                  RelativeLists
                  TenThousandth}
{RelativeTenThousandth {RelativeSquareRoot 1.0 TenThousandth 2.0}
                       '~=~' 1.41421356}
{RelativeTenThousandth {RelativeSquareRoot 1.0 TenThousandth 4.0}
                        '~=~' 2.0}
{RelativeTenThousandth {RelativeSquareRoot 1.0 TenThousandth 64.0}
                       '~=~' 8.0}
{RelativeTenThousandth {RelativeSquareRoot 1.0 TenThousandth 3.14159}
                       '~=~' 1.7724531}
{RelativeTenThousandth {RelativeSquareRoot 1.0 TenThousandth 9.0e24}
                       '~=~' 3.0e12}
{RelativeTenThousandth {RelativeSquareRoot 1.0 TenThousandth 9.0e~40}
                       '~=~' 3.0e~20}
```

Figure 8: Tests for Exercise 7 on page 5.

```
\insert 'DiffApproxims.oz'
\insert 'FloatTest.oz'
{StartTesting 'DiffApproxims...'}
{WithinTest {List.take {DiffApproxims 500.0 fun {$ X} X*X end 20.0} 9}
 '~=~' [540.0 290.0 165.0 102.5 71.25 55.625 47.8125 43.9062 41.9531]}
{WithinTest {List.take {DiffApproxims 100.0 fun {$ X} X*X*X end 10.0} 8}
 '~=~' [13300.0 4300.0 1675.0 831.25 526.562 403.516 349.316 324.048]}
```

Figure 9: Tests for Exercise 8 on page 5.

```
Differentiate: <fun {$ Float Float  <fun {$ Float}: Float> Float}:
                    Float>
```

such that `{Differentiate Epsilon Delta0 F N}` returns an approximation that is accurate to within `Epsilon` to the derivative of `F` at `N`. Use the previous problem (Exercise 8 on page 5) with `Delta0` as the initial value for `Delta`. Examples are given in Figure 10 on the following page.

## Textbook Problems

10. (20 points) [Thread Semantics] Do problem 1 in section 4.11 of the textbook [RH04]. Note that part (b) should read (according to the errata) "All of these executions . . .".

11. Do one of the following problems.

    (a) (10 points) [Threads and garbage collection] Do problem 2 in section 4.11 of the textbook [RH04].

    (b) (10 points) [Order-determining concurrency] Do problem 4 in section 4.11 of the textbook [RH04].

    (c) (10 points) [The Wait operation] Do problem 5 in section 4.11 of the textbook [RH04].

12. (20 points) [Dataflow behavior in a concurrent setting] Do problem 8 in section 4.11 of the textbook [RH04].

13. (0 points) [Digital logic simulation] This problem was withdrawn, as the solution already appears in the text.

14. Do one of the following.

    (a) (10 points) [Laziness and incrementality] Do problem 12 in section 4.11 of the textbook [RH04].

    (b) (10 points) [Laziness and monolithic functions] Do problem 13 in section 4.11 of the textbook [RH04].

15. (20 points) [By-need execution] Do problem 16 in section 4.11 of the textbook [RH04].

16. Do one of the following.

    (a) (15 points) [Concurrency and Exceptions] Do problem 18 in section 4.11 of the textbook [RH04].

    (b) (15 points) [Limitations of Declarative Concurrency] Do problem 19 in section 4.11 of the textbook [RH04].

## Extra Credit Problems

17. Do any of the textbook problems you didn't choose to do above. Clearly label that these problems are being done for extra credit.

# References

[RH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, Cambridge, Mass., 2004.

```
\insert 'Differentiate.oz'
\insert 'FloatTest.oz'
{StartTesting 'Differentiate...'}
Millionth = 0.0000001
WithinMillionth = {FloatTestMaker
                   Within
                   WithinLists
                   Millionth}
{WithinMillionth {Differentiate Millionth 500.0 fun {$ X} X*X end 20.0}
 '~=~' 40.0}
{WithinMillionth {Differentiate Millionth 100.0 fun {$ X} X*X*X end 10.0}
 '~=~' 300.0}
```

Figure 10: Tests for Exercise 9 on page 5.